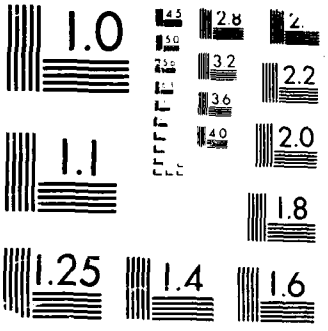
AD-A194 566 CONCURRENT COMPUTER ARCHITECTURE(U) MASSACHUSETTS INST 1/1
OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB
W J DALLY NOV 87 VLSI-MEMO-87-422 N00014-80-C-0622
UNCLASSIFIED F/G 12/6 NL



MICROCOPY RESOLUTION TEST CHART

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

VLSI Memo No. 87-422
November 1987

CONCURRENT COMPUTER ARCHITECTURE

William J. Dally

Abstract

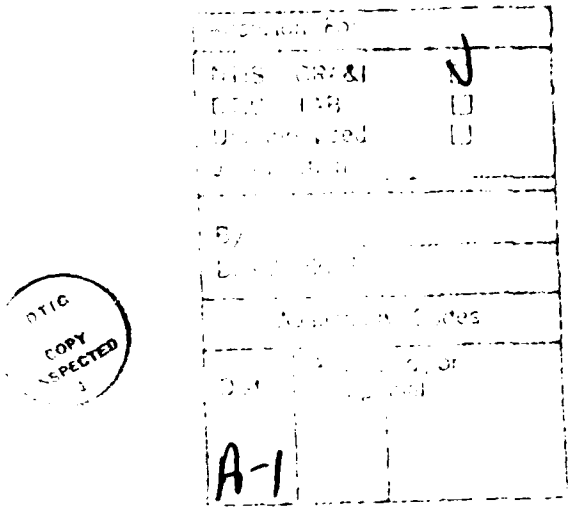
Present generation concurrent computers offer performance greater than vector supercomputers and are easily programmed by non-experts. Evolution of VLSI technology and a better understanding of concurrent machine organization have led to substantial improvements in the performance of numerical processors, symbolic processors, and communication networks. A 100MFLOPS arithmetic chip and a 5μs latency communication network are under construction. Low-latency communication and task switching simplify concurrent programming by removing considerations of grain size and locality. A message-passing concurrent computer with a global virtual address space provides programmers with both a shared memory, and message-based communication and synchronization. This paper describes recent advances in concurrent computer architecture drawing on examples from the J-Machine, and experimental concurrent computer under development at MIT.

Microsystems Research Center
Room 39-321

Massachusetts Institute of Technology

Cambridge Massachusetts 02139

Telephone (617) 253-8138



Acknowledgements

This work was supported in part by the Defense Advanced Research Projects Agency under contract numbers N00014-80-C-0622 and N00014-85-K-0124, and by a National Science Foundation Presidential Young Investigators Award, with matching funds from the General Electric Corporation.

Author Information

Dally: Artificial Intelligence Laboratory and the Laboratory for Computer Science, MIT, Room 43-419, Cambridge, MA 02139, (617)253-6043.

Concurrent Computer Architecture[1]

William J. Dally
Artificial Intelligence Laboratory and
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

ABSTRACT

Present generation concurrent computers offer performance greater than vector supercomputers and are easily programmed by non-experts. Evolution of VLSI technology and a better understanding of concurrent machine organization have led to substantial improvements in the performance of numerical processors, symbolic processors, and communication networks. A 100MFLOPS arithmetic chip and a $5\mu s$ latency communication network are under construction. Low-latency communication and task switching simplify concurrent programming by removing considerations of grain size and locality. A message-passing concurrent computer with a global virtual address space provides programmers with both a shared memory, and message-based communication and synchronization. This paper describes recent advances in concurrent computer architecture drawing on examples from the J-Machine [13], an experimental concurrent computer under development at MIT.

[1]The research described in this paper was supported in part by the Defense Advanced Research Projects Agency under contracts N00014-80-C-0622 and N00014-85-K-0124 and in part by a National Science Foundation Presidential Young Investigator Award with matching funds from General Electric Corporation.

NOMENCLATURE

The following symbols are used in this paper. Where a symbol has been overloaded, its meaning is clear from context.

A	chip area (m^2),
a	acceleration (m/s^2),
b	a body,
C	capacitance (F),
C	set of channels,
D	distance (hops),
f	frequency (1/s),
g	gravitational constant, $6.67 \times 10^{-}11$ (Nm^2/kg^2),
k	radix - the number of nodes along each dimension of a cube,
L	message length (bits)
m	mass (kg),
N	number of bodies,
n	dimension - the number of dimensions of a cube,
N	set of nodes,
P	power (W),
T	total time,
T_C	channel cycle time (s),
T_P	node compute time (s),
T_{SF}	store-and-forward routing latency (s),
T_{WH}	wormhole routing latency (s),
V	voltage (V),
v	velocity (m/s),
W	channel width (bits),
x	position (m),

1. INTRODUCTION

Present generation concurrent computers offer performance greater than vector supercomputers and are easily programmed by non-experts.

Evolution of VLSI technology and a better understanding of concurrent machine organization have led to substantial improvements in the performance of numerical processors, symbolic processors, and communication networks. For example, VLSI technology makes it possible to build a single-chip 100MFLOPS numerical processor. Advances in machine organization allow us to configure this processor in a manner that reduces its I/O bandwidth requirements to a level that can be handled by a communication network. A machine with 2K such numerical processors fits on 16 printed-circuit boards and gives a peak performance of 200GFLOPS. Usable performance is estimated at 2GFLOPS.

Low-latency communication networks, and message-driven processor architectures simplify concurrent programming by removing considerations of locality. The use of low-dimensional networks and efficient routing algorithms allow us to send a 6-word (216-bit) message across the diameter of a 4K node concurrent computer in 5μs (first bit out to last bit in). A message-driven processor can perform a task switch in response to this message in 1μs. This low-latency communication makes program performance largely independent of how code and data are partitioned and placed on the nodes of a concurrent computer. Programmers need not be concerned with the topology or size of the machine they are programming.

A message-passing concurrent computer with a global virtual address space combined with low-latency networks and processors provides users with the best features of both message-passing

and shared-memory architectures. Such a machine supports the fast local memory access and message-based synchronization/communication of a message-passing machine. At the same time it provides the uniform naming of objects, position independence, and ability to share code and data characteristic of a shared-memory machine. This convergence of two families of parallel computers results in a machine that efficiently supports a broad range of concurrent programming models.

This paper describes recent advances in concurrent computer architecture drawing on examples from the J-Machine [13], an experimental concurrent computer under development in the Concurrent VLSI Architecture Group at MIT. Section 2 discusses object-oriented concurrent programming. An example program for simulating the dynamics of an N-body problem is presented. Low-latency interconnection networks are described in Section 3. It is shown that low-dimensional networks outperform binary n-cubes (hypercubes). To exploit the low-latency of these networks requires processing elements that can react quickly to the arrival of messages. The architecture of such a message-driven processor is described in Section 4. To solve problems in the physical sciences requires numerical performance. Progress towards a 100MFLOPS numerical processor chip is described in Section 5.

2. CONCURRENT PROGRAMMING

The Problem

Concurrent programming is often considered harder than sequential programming because of partitioning, communication, and synchronization. If a machine is programmed at a very low level, concurrent programming can indeed be a difficult task, and the programs produced are rarely portable. However, with suitable programming abstractions [12], concurrent programming need be no harder than sequential programming. In this section I discuss some issues in concurrent programming and work an example using an object-oriented concurrent programming language currently under development at MIT. The hardware architecture described in the remainder of the paper has been developed to efficiently support this style of programming.

In a concurrent program, the data must be partitioned over the nodes of a concurrent machine and the program must be partitioned into tasks that are scheduled separately and possibly concurrently. Most problems, however, have a natural partitioning. All programmers partition their data into objects (e.g., records in Pascal [35] or structures in C [20]) and their programs into methods or procedures. If our architecture and programming system allow us to exploit this natural partition, then partitioning is not a difficult task. The natural partition of most programs tends to be fine grained with objects averaging 8 words and methods averaging 20 instructions [6]. To efficiently support the natural partition of most programs, an architecture and programming system must support fine-grained concurrency.

The objects (data partitions) must be placed on the nodes of a concurrent computer. In the past, a great deal of attention was paid to developing placements that took advantage of the underlying machine topology [5]. Careful placement was necessary because of the slow communication and non-uniform addressing of early concurrent computers [31]. However, the development of message passing machines with fast communications (Section 3) and a uniform global address space (Section 4) has made placement less of an issue. In such machines a random placement performs nearly as well as an optimum placement, and an initial placement may be improved at runtime by object migration.

A concurrent program specifies the communication of data between partitions. In some concurrent programming systems, communication is made difficult by non-uniform naming: local objects are referenced differently than non-local objects. In the Cosmic Kernel [34], for example, local objects may be referenced through a pointer, while global objects require an explicit message send and receive. Providing a global address space allows objects to be referenced via a

single mechanism (the virtual address) regardless of their location, and relieves the programmer of the bookkeeping required to keep track of node numbers. Programs become both easier to write and more portable.

Synchronization is required to schedule the tasks of a concurrent program in an order that assures correct results. In the example below, a task that computes the acceleration of a body, a_i, must be completed before a computation that integrates the acceleration to update the velocity, v_i, is run. In another program [5], a task that updates a record in a data structure must exclude other tasks from accessing the record until the update is complete.

In a *message-passing* programming system [6] [1] [2], a computation is performed by sending a message to an object (e.g., sending a_i to a body). The message serves both to communicate the required data and to schedule the task that operates on the data (e.g., computing v_i). This message-driven scheduling requires us to create a new process for each message received. On a conventional processor, this would be a prohibitively costly operation. A processor architecture tailored for this model of computation (Section 4), however, can perform the required process creation and switching efficiently. Having a single, general method of synchronization, process creation on message arrival, allows us to avoid more restricted forms of synchronization such as barriers that lead to serial bottlenecks.

Message passing results in a good placement of tasks. By scheduling the operation to be performed on the node where the body is located, the position and previous velocity of the body can be accessed without further communication. Communication in a message-passing system is also more efficient than in a shared-memory system [17] [3] since the communication overhead is paid only once for an entire logical message (e.g., acceleration vector) rather than being paid for each word transferred.

It is possible to recode an application to run on a concurrent computer without abandoning all of the existing code. Approaches that attempt to translate *dusty* FORTRAN decks into concurrent programs by sophisticated compilation techniques [23] [26] are limited to speedups of ≈ 30 because the algorithm being translated is often inherently sequential even though the problem it is solving is not. A different approach is to recode only those parts of the program that are affected by concurrency. In a typical switch-level circuit simulation program [4], the majority of the code comprised user interface, file I/O, and model equation routines that are not changed in a concurrent implementation. Only those portions of the program that involve the *system-level* solution must be rewritten. In the simulator these portions, the event scheduler and path finder, comprised less than 10% of the code.

An Example

Consider the gravitational N-body problem: given N bodies in space with initial positions and velocities, compute their trajectories for a given time interval assuming that they interact only through gravitational attraction. We will develop a program to solve this problem in Concurrent Smalltalk (CST) [6], a concurrent programming language based on Smalltalk-80 [16]. A description of the programming language is beyond the scope of this paper.

The natural partition for the N-body problem is to define:

- an object for each body,
- a method to compute the accelerations due to the interaction of one pair of bodies[2],
- a method for integrating the acceleration on one body to compute its new position and velocity.

[2]We compute acceleration rather than force to correctly handle the case of zero-mass bodies.

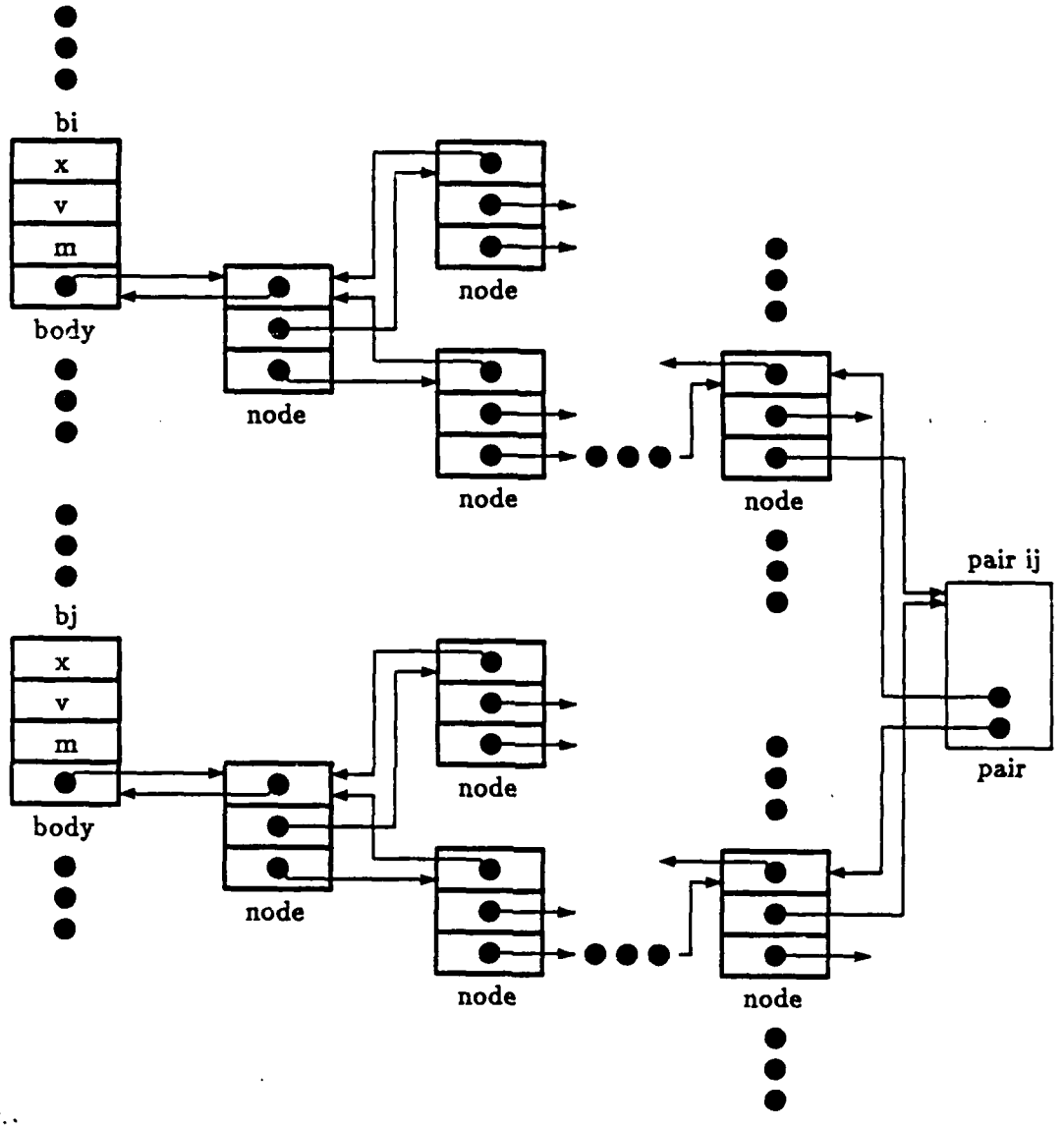


Figure 1: Data structure for the N-body problem. The structure distributes the position and mass of each body, b_i, to all pairs containing b_i. The structure then sums the accelerations on b_i due to each pair containing b_i.

To connect up these partitions, we need a data structure, as shown in Figure 1, that distributes the positions of the bodies to form all unique pairs $(b_i, b_j), i < j$, and sums the accelerations acting on each body to compute $a_i = \sum_{j \neq i} a_{ij}$, where a_{ij} is the acceleration on one body, b_i, due to another body, b_j.

As shown in Figure 2, we declare three classes of objects: bodies, nodes, and pairs to build the data structure shown in Figure 1. Each declaration specifies a class by listing its superclass and its instance variables. In this example, the superclass of each class is `Object`, the root of the class hierarchy. Comments are enclosed in double quotes, ".

```
"A body has position, x, velocity, v, and mass m.  x and v are 3-vectors.
 node is the connection from the body to the data structure and
 i is the index of the body"
Class Body (Object) | x v m node i t.

"A node in the data structure connects its parent to its two children.
 The node's state records data that has been received from one child
 while waiting for data from the other child."
Class Node (Object) | parent lchild uchild state.

"A pair forms a leaf of the data structure, synchronizing the data required
 for the acceleration calculation
 i, j - body indices, nodei, nodej - connections to data structure;
 mi, mj -  masses, xi, xj - positions (3-vectors), state - number received"
Class Pair (Object) | i j nodei nodej mi mj xi xj state.
```

Figure 2: Class declarations for the N-body problem. Objects of class body hold the state of each body, objects of class node form the internal nodes of the data structure, and objects of class pair form the leaves of the data structure.

Using this data structure, we compute the trajectories as follows:

1. Create and initialize an object for each body.
2. Build the data structure shown in Figure 1.
3. For each time step.
   (a) Each body sends its position and mass through the data structure to all relevant pairs.
   (b) Each pair (b_i, b_j) computes a_{ij} and a_{ji}. We compute all $N(N-1)/2$ interactions in parallel.
   (c) Each pair sends its results through the data structure back to the bodies. The data structure sums the partial accelerations to compute the total acceleration on each body.
   (d) Each body integrates its acceleration to update its velocity and position[3].

The computation is globally asynchronous, so execution of the different steps of this algorithm may overlap. All synchronization is performed locally by the arrival of data. For example, at a given point in time, some bodies may be in step 3a, others in step 3b, and others in step 3c.

Assume that we have already created and initialized a distributed collection of N objects, and a distributed collection of $N(N-1)/2$ pairs. Figure 3 shows how each body constructs a fanout/combine tree of nodes (Figure 1). Each body first creates a root node. The body then sends a message to the root asking to be connected to a range that includes all other nodes. The tree is then recursively constructed by subdividing the range of nodes for which connections are to be made.

[3]Some integration methods may require several acceleration computations per timestep.

```
"Build a tree of nodes that connects a body to all relevant pairs
 pairs is a distributed collection of N(N-1)/2 pairs"
(Body) build: pairs
  node <- Node new.
  node connect: self nr: i from:0 to: n with: pairs.

"Computes the index of the pair for nodes i and j"
(Integer) pairIndex: j
  self < j ifTrue: [^ (j*j + j)//2 + self]
           ifFalse: [^ (self*self + self)//2 + j].

"Recursively expand a node of the fanout/combine tree
 There is no check for i=j, instead diagonal pairs always return a=0"
(Node) connect: p nr: i from: l to: u with: pairs
  | mid |
  parent <- p.
  mid <- l + u // 2.
  l = mid ifTrue:  [lchild <- pairs at: (i pairIndex: l).
                    lchild connect: self nr: i]
          ifFalse: [lchild <- Node new.
                    lchild connect: self nr: i from: l to: mid with: pairs].
  u = (mid+1) ifTrue:  [uchild <- pairs at: (i pairIndex: u).
                        uchild connect: self nr: i]
              ifFalse: [uchild <- Node new.
                        uchild connect: self nr: i from: mid+1 to: u with: pairs].

"End the recursion by connecting a pair to the data structure"
(Pair) connect: p nr: index
  index = i ifTrue: [nodei <- p]
            ifFalse: [nodej <- p].
```

Figure 3: The fanout/combine tree from a node to all related pairs is recursively constructed by sending the `connect:...` message to a node.

Once the data structure is created, a timestep is simulated by sending each body a step message. As shown in Figure 4, the body sends its position, mass, and index to the root node of its fanout tree. The tree distributes this data to all related pairs without waiting for any replies. After a pair has received data from both of its nodes, it calculates the accelerations using (1) and sends the results up the data structure. Each node adds the accelerations received from its two children and forwards the sum up the tree. When the body receives the total acceleration, it integrates to compute its new velocity and position.

$$\mathbf{a}_{ij} = \left(\frac{-gm_j}{|\mathbf{x}_i - \mathbf{x}_j|^2}\right)\left(\frac{\mathbf{x}_i - \mathbf{x}_j}{|\mathbf{x}_i - \mathbf{x}_j|}\right). \tag{1}$$

```
"Simulate motion of body for time interval dt"
(Body) step: dt
  t <- dt.
  node pos: x mass: m index: i.

"Fanout position, x, and mass, m, for body number i"
(Node) pos: x mass: m index: i
  state <- nil.
  lchild pos: x mass: m index: i.
  rchild pos: x mass: m index: i.

"When position and mass are received from both bodies i and j
 compute accelerations and forward down trees"
(Pair) pos: x mass: m index: index
  |aij aji diff dist tmp|
  index = i ifTrue: [xi <- x. mi <- m]
            ifFalse: [xj <- x. mj <- m].
  state <- state + 1.
  state = 2 ifTrue: [ diff <- xi - xj.
                      dist <- diff abs.
                      tmp <- (g * diff) / (dist * dist * dist).
                      aij <- tmp negated * mj.
                      aji <- tmp * mi.
                      nodei accel: aij.
                      nodej accel: aji.
                      state <- 0].

"Each node adds the accelerations from its two children and
 forwards the sum to its parent"
(Node) accel: a
  state isNil ifTrue: [state <- a]
              ifFalse: [parent accel: a+state].

"When a body receives its total acceleration, it integrates to
 compute its velocity and position"
(Body) accel: a
  v <- v + a*t.
  x <- x + v*t.
```

Figure 4: A time step is simulated by distributing the bodies' positions to all pairs, computing the accelerations, and integrating.

For the sake of clarity, this example has glossed over a number of subtle points:

sending by value: Smalltalk normally sends compound objects (viz. objects that do not fit into a machine word) by reference. In this example, the position vector `x` is a compound object. The program as presented will work, but to improve execution efficiency we would like to send `x` by value. Concurrent Smalltalk includes a language feature to force an object to be sent by value. Its use has been omitted here.

distribution details: The code presented here works only for an even number of bodies, and the handling of the diagonal pairs has been avoided. The addition of some conditional code fixes these problems.

placement: Since no placement information is specified, the objects will be placed randomly. Execution efficiency can be improved by refining the placement of the fanout/combine trees to exploit locality. This task can be performed by an iterative improvement algorithm based on simulated annealing [21] [33]. However, the evolution of high-performance interconnection networks (Section 3) is making placement less of an issue.

integration: In practice a higher-order integration method with adaptive stepsize control would be used [27].

Concurrent programming is not difficult. The majority of the code in this example is identical to a sequential program and is concerned with the physics of the problem (e.g., the acceleration equation), or the simulation method (e.g., the integration formula). The programming style presented here isolates the programmer from the details of the hardware. This code is transportable across machines with different numbers of nodes, different communication topologies, and different node types. The programmer expends no effort on partitioning. The natural partition of the problem is used. A minimum of effort is spent on synchronization. Most synchronization is by message passing. The major complication faced by the programmer is the construction of the fanout/combine tree for each node (Figures 2 and 3). At MIT we are working on the development of abstractions that will largely automate this part of concurrent programming.

The high performance offered by concurrent computation is a strong inducement to learn this new programming style. This solution to the N-body problem has an asymptotic speedup of $O(N^2/\log N)$ over its sequential counterpart. The time required on a sequential computer is $O(N^2)$, while the time required on a concurrent machine is $O(\log N)$. The log factor is due to the time required by the fanout/combine trees to distribute the positions and accumulate the accelerations. By exploiting concurrency within equations (Section 5) an additional constant factor of speedup is attained.

A 4K node J-machine, which will fit on 16 400mm square PC boards, will compute one timestep of a 100-body problem in $\approx 50\mu$s[4]. Each timestep requires $\approx 1.2 \times 10^5$ floating-point calculations for a throughput of 2.4GFLOPS. This represents a speedup of ≈ 200 over a 10MFLOPS mainframe, and few mainframes will achieve 10MFLOPS on such a problem.

To achieve this level of performance, a concurrent computer must efficiently execute fine grain programs with a minimum of overhead. Of the 50μs per timestep, only 1.8μs is required to perform an acceleration computation, and with 2K RAP chips (Section 5), all acceleration computations are computed in 9μs. Much of the remaining 41μs is taken up by communication and synchronization overhead. On the Cosmic Cube [31], this overhead would be 10ms. The remainder of this paper discusses recent advances in concurrent computer architecture (fast networks, message-driven processors, and specialized arithmetic processors) that have reduced this overhead to a reasonable level.

[4]This number has been estimated by counting instructions and communication delays.

3. INTERCONNECTION NETWORKS

Efficient execution of fine grain concurrent programs requires a fast communication network. Consider the computation graph of Figure 5. The time required to solve a problem on a concurrent computer is bounded below by the sum of the communication time and the computation time along the critical path of the computation. Benchmark studies of a number of fine-grain concurrent programs show that each local computation (vertex) typically involves executing 20 instructions (requiring a node compute time, T_P, of about $5\mu s$). To support fine-grain concurrent computation, the network must have a latency (between any two nodes)$\leq T_P$ and be able to handle a message from each processing node each T_P.

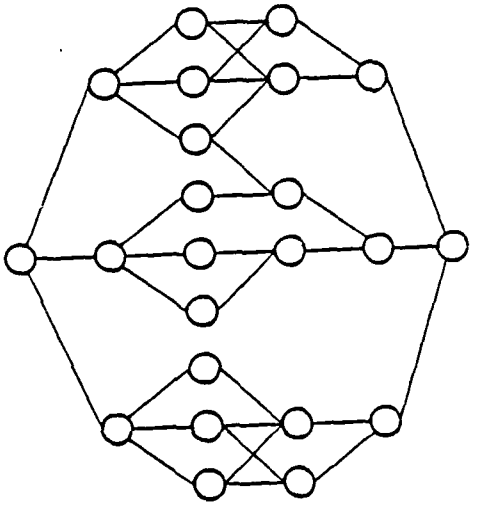


Figure 5: The computation graph of a concurrent program. The vertices represent a local computation being performed at a node of a concurrent computer. The edges represent communication actions between nodes. The time required to perform the computation is bounded below by the sum of edge and vertex times along the critical path for the computation.

Existing networks fall short of this required performance. In machines employing these networks, several *grains* of computation must be grouped together to avoid excessive overhead. The result is a reduction in concurrency. By reducing network latency, we linearly increase the amount of concurrency we are able to exploit (for a given overhead). For example, if an iPSC [18] with 10ms latency can exploit 100-fold concurrency on a problem with a fine grain size, a J-Machine with $5\mu s$ latency can exploit 200,000-fold concurrency on the same problem.

VLSI systems are wire limited. The cost of these systems is predominantly that of connecting devices, and the performance is limited by the delay of these interconnections. Thus, an interconnection network must make efficient use of the available wire. The topology of the network must map into the three physical dimensions so that messages are not required to *double back* on themselves, and in a way that allows messages to use all of the available bandwidth along their path. Also, the topology and routing algorithm must be simple so the network switches will be sufficiently fast to avoid leaving the wires idle while making routing decisions.

Our recent findings suggest that low-dimensional k-ary n-cube interconnection networks [8] using *wormhole routing* [30] [19] and *virtual channels* [9] are capable of providing the performance required by fine-grain concurrent architectures. To test these ideas, we have constructed a prototype VLSI routing chip, the torus routing chip (TRC) [7], and are in the process of designing a second chip, the network design frame (NDF) [11].

Wormhole Routing

With *wormhole routing* (Figure 6B) as soon as each *flit* (flow-control digit) of a message arrives at a node it is forwarded to the next node. With *store-and-forward routing* (Figure 6A), the method

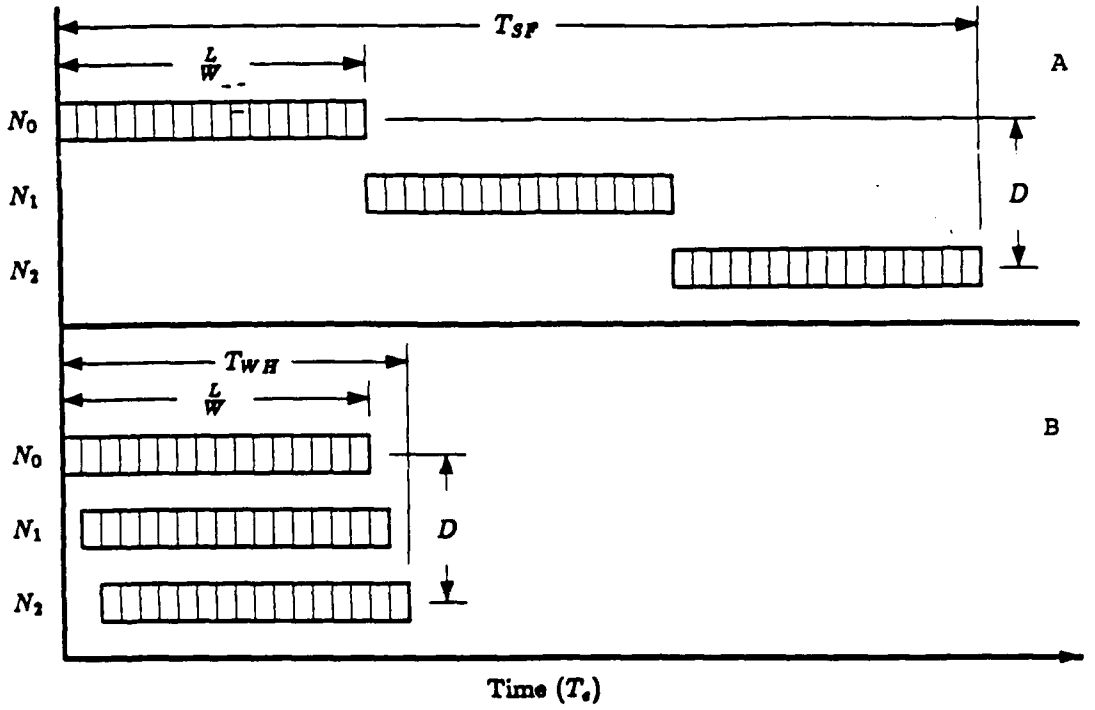


Figure 6: The latency of *store-and-forward routing* (A) compared to *wormhole routing* (B). Wormhole routing reduces latency from the product of $\frac{L}{W}$ and D to the sum of these two components.

used by most existing concurrent computers, the entire message is received before forwarding the packet to the next node. Using wormhole routing gives a network latency, T_{WH}, that is the sum of a component due to message length normalized to channel width $\frac{L}{W}$, and a component due to the distance the message must travel, D. With *store-and-forward* routing, on the other hand, the latency, T_{SF}, is the product of these two components.

$$T_{WH} = T_C \left(\frac{L}{W} + D \right), \tag{2}$$

$$T_{SF} = T_C \left(\frac{L}{W} \times D \right), \tag{3}$$

where T_C is the channel transmission time, L is the message length in bits, W is the channel width in bits, and D is the number of channels the message must traverse (distance).

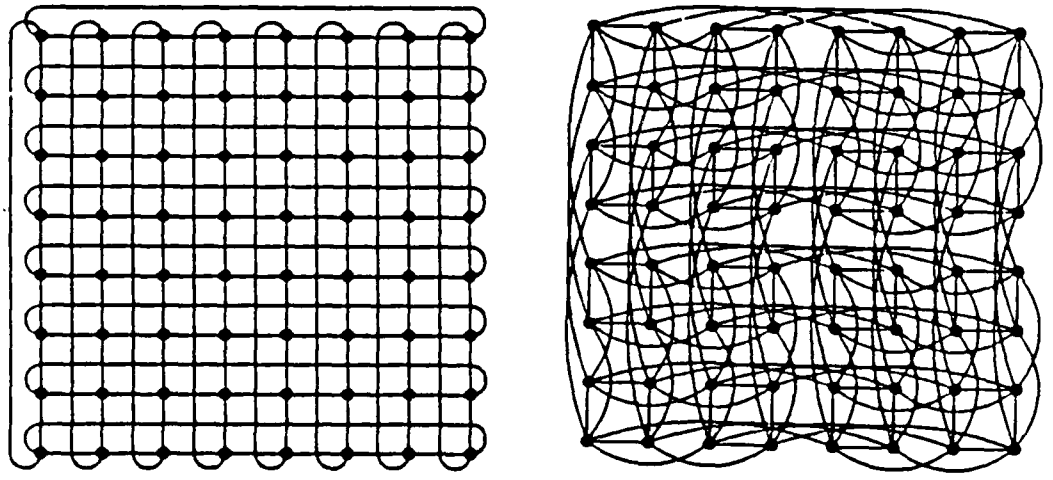
Consider a concurrent computer with 64K nodes connected as a 16-ary 4-cube with 8-bit wide channels ($W = 8$). Assuming no locality, the average distance a message must travel in this machine is $D = 15$. For 256-bit messages, $T_{WH} = 47T_C$, an order of magnitude less than $T_{SF} = 480T_C$.

Low-Dimensional Cubes

Many concurrent computers have been built using binary n-cube (hypercube) interconnection networks because these networks are optimal when all channels are considered equal. However, considering a channel in a binary n-cube to be equal to a channel in a low-dimensional network is not a reasonable assumption. Because binary n-cubes have long wires and high bisection widths their channels are typically narrower and slower than the channels in a low-dimensional network. When these factors are taken into account, the low-dimensional networks out-perform the high-dimensional networks.

Consider the networks shown in Figure 7. Suppose the binary 6-cube has 4-bit wide channels (as in the Caltech Cosmic Cube [31]). An 8-ary 2-cube with 16-bit wide channels has the same wiring complexity. With wormhole routing and 256-bit messages the 6-cube has a latency of $67T_C$ while the 2-cube has a latency of only $20T_C$. Increasing the radix, k, of a k-ary n-cube while holding wiring complexity (bisection width) constant increases both $W \propto k$ and $D \propto kn$. This decreases the component of latency due to message length, $\frac{L}{W}$, while increasing the component due to distance, D. The minimum latency occurs when these two components are nearly equal (Figure 8). For $L \approx 200$ the optimum dimension, n, is two for up to 1K nodes and three for 1K to 32K nodes, and four for 32K to 1M nodes.

The throughput of a network is the maximum number of messages that can be delivered per unit time. It is often expressed as a fraction of the network's capacity, the number of messages that would be delivered if every channel of the network was fully used. As the amount of traffic in the network increases, the latency of a message is increased. The latency given by (2) assumes an unloaded network.



A B

Figure 7: Two 64-node k-ary n-cubes: an 8-ary 2-cube (A) and a binary 6-cube (B). Network A has a bisection width of 16 channels while B has a bisection width of 64 channels. Thus the channels in A can be made four times as wide as the channels in B for the same wiring complexity.

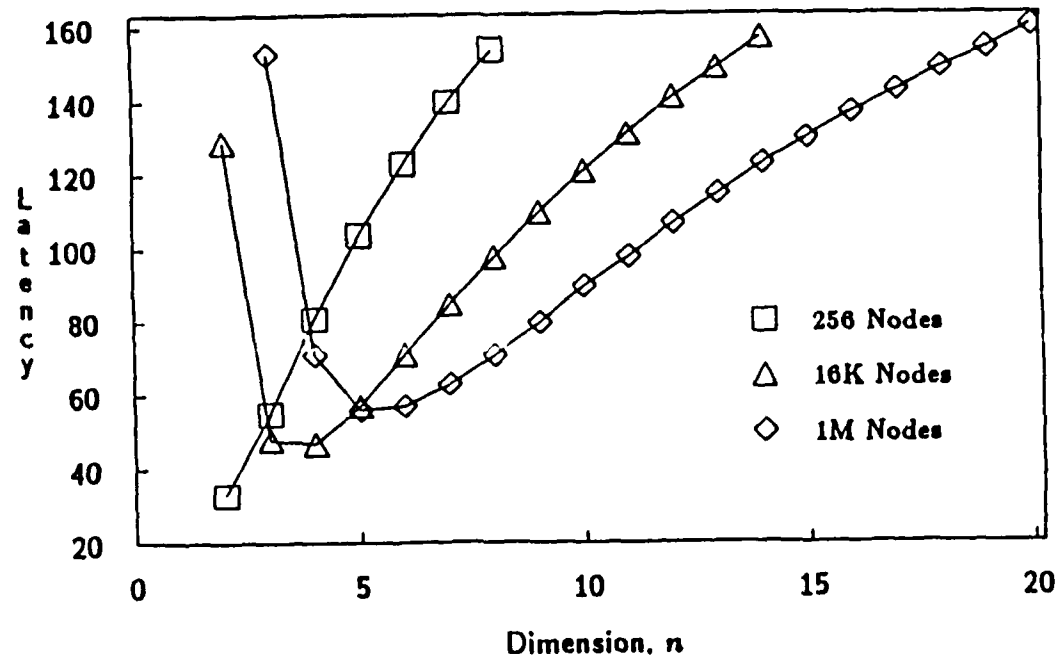


Figure 8: Latency as a function of dimension for networks of constant bisection width (B=N, L=150). Low-dimensional networks (left) are distance limited, while high-dimensional networks (right) are message-length limited.

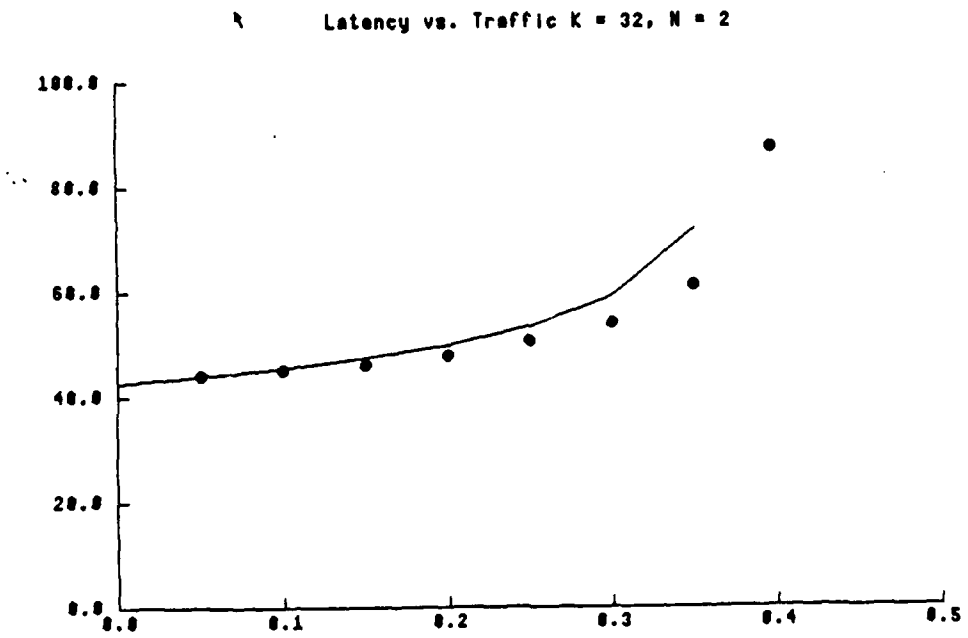


Figure 9: Latency vs. Traffic for a 32-ary 2-cube, L=200bits. Solid line is predicted latency, points are measurements taken from a simulator.

We have recently developed a queueing model of k-ary n-cube wormhole networks that accurately predicts the latency as a function of network traffic, and allows us to calculate the maximum throughput for a given network configuration [8]. Figure 9 shows how latency varies with traffic for a 32-ary 2-cube (1024 nodes). The solid line is the predicted latency. The points are measurements taken from a simulator. The model agrees with the simulation within 5%, with the model being slightly pessimistic, until the network approaches saturation. Latency increases less than 20% as traffic is increased from zero to 30% capacity. Saturation (maximum throughput) occurs at $\approx 40\%$ capacity.

Low-dimensional networks have several other advantages.

- Because wires are shorter, the channels in these networks typically operate faster than in high dimensional networks, increasing throughput and further decreasing latency.
- Low-dimensional networks have better queueing performance. If one thinks of channels as being servers, these networks have fewer servers with greater capacity resulting in a lower average service time.
- Because the control logic for a network switch typically scales with the number of dimensions, the switches for low-dimensional networks are simpler than those for high-dimensional networks.

Virtual Channels

Until recently there was no known algorithm for deadlock-free routing in k-ary n-cube, wormhole networks. The conventional *structured buffer pool* algorithms that are used in store-and-forward networks are not applicable to networks that use wormhole routing. These algorithms interleave the items being buffered (packets in a store-and-forward network), but wormhole networks buffer flits that cannot be interleaved.

We have recently developed a new class of algorithms for deadlock free routing based on the concept of *virtual channels*. Shown in Figure 10, virtual channel algorithms operate by restricting routing rather than by restricting buffer allocation. To do this requires that routing be a function of the channel a message arrives on and the destination node, $C \times N \mapsto C$, rather than the node a message is on and the destination node, $N \times N \mapsto C$. Projecting this function gives a dependency relation among channels. By multiplexing several virtual channels on each physical channel we can restrict routing in a manner that avoids deadlock without loosing strong connectivity. A set of virtual channels all share the same physical wires. Each virtual channel requires only a single flit buffer. The virtual channel method can be used to route deadlock free in any strongly connected network [9].

The Torus Routing Chip

The Torus Routing Chip (TRC), shown in Figure 11, is a self-timed [29] VLSI chip that performs wormhole routing in k-ary n-cube networks, and uses virtual channels to prevent deadlock [7]. A single TRC provides 8-bit data channels in two dimensions and can be cascaded to add more dimensions. A TRC network can deliver a 150-bit message in a 1024 node 32-ary 2-cube with an average latency of 7.5μs.

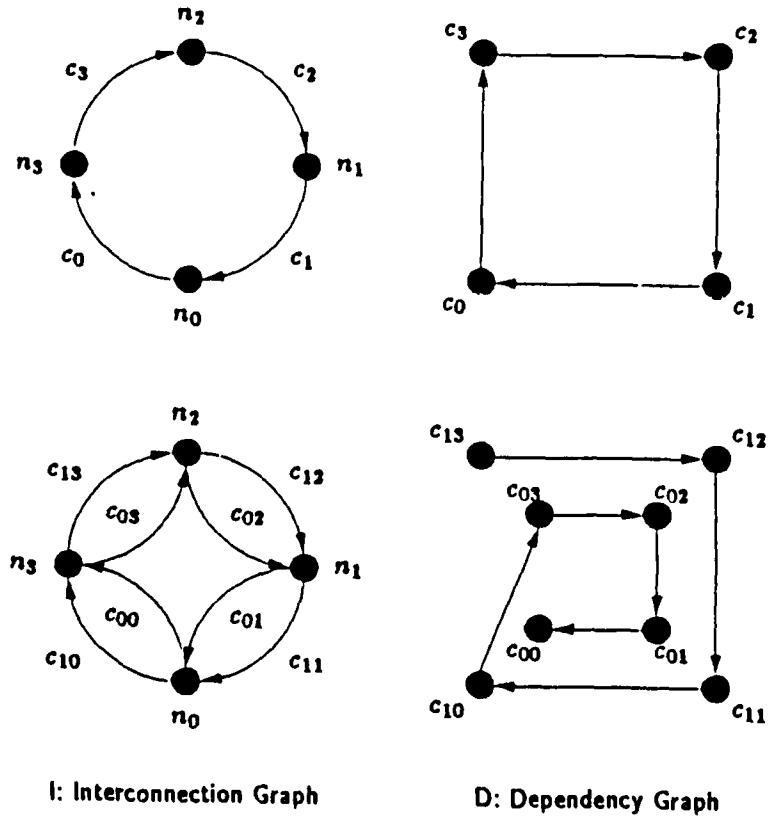


Figure 10: Considering routing to be a function $C \times N \mapsto C$ rather than the conventional $N \times N \mapsto C$ deadlock corresponds to cycles in the channel dependency graph (right) rather than the interconnection graph (left). By multiplexing two virtual channels on each physical channel, we can restrict the routing function to eliminate deadlock (bottom).

The Network Design Frame

We have recently undertaken the design of the Network Design Frame (NDF) [11] which we plan to use for the interconnection network of the J-Machine. The circuitry of the NDF will be integrated into the pad-frame of a VLSI chip. The center of the chip is left uncommitted to be used for the circuitry of a network node. We plan to use the NDF to perform communication for the MDP (Section 4) and the RAP (Section 5).

The NDF incorporates a partitioned switch architecture, bidirectional data channels, and low-voltage output drivers to achieve a worst-case latency of 5μs in a 4K node 64-ary 2-cube. In the partitioned switch architecture, shown in Figure 12, the routing logic is partitioned into two-way switches. The partitioned switch's data paths and control logic are simpler (and thus smaller and faster) than the centralized crossbar design used in the TRC. A signal passes through only 10 gate delays from input to output for a propagation delay of 20ns (estimated).

Bidirectional data channels are used in the NDF to reduce latency and to exploit locality.

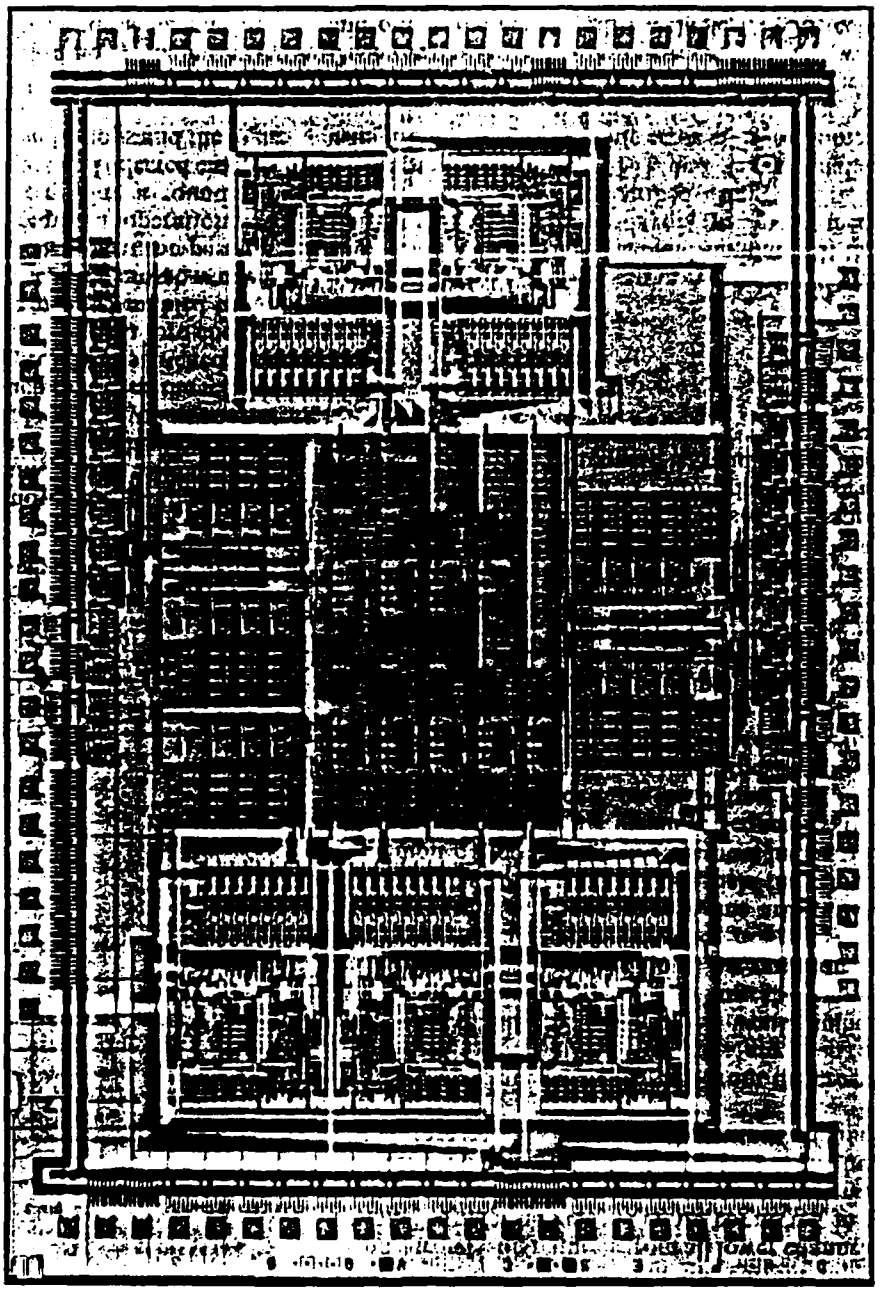


Figure 11: Photomicrograph of the Torus Routing Chip (TRC).

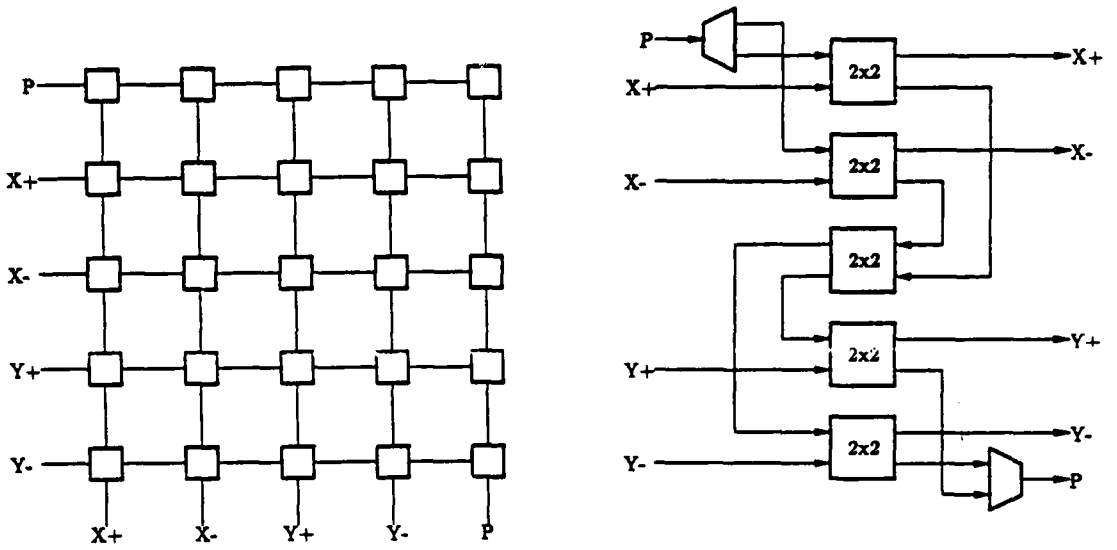


Figure 12: By using a partitioned datapath (right) the NDF requires less area and runs faster than the TRC which uses a centralized crossbar switch (left).

Because wire density is a major limitation, the two directions of communication will share the same data wires. While the NDF is constructed using CMOS technology, communication on these bidirectional data wires uses ECL signal levels to improve speed, reduce power dissipation, and reduce noise. The NDF uses low-voltage swing output pads based on a design by Tom Knight [22]. Reducing the voltage swing by a factor of 5 makes these pads 5 times as fast as conventional pads. Also, because power goes as the square of voltage, $P = CV^2f$, these pads dissipate 1/25 (4%) as much power as conventional pads. Since much of the power in the machine goes into driving the inter-node wires, this savings represents a considerable reduction in total power dissipation.

4. SYMBOLIC PROCESSING ELEMENTS

To support the fine-grained concurrent programming model discussed in Section 2, a processing element must:

- perform rapid context (task) switching,
- perform rapid message handling (buffering and scheduling),
- support a global address space.

Conventional instruction processors are ill-suited to serve as processing nodes in a concurrent computer. Their I/O systems are designed to handle high-latency peripherals (e.g., disks) and thus they respond slowly ($\approx$ 100 instruction times) to messages arriving over the network. Also, their register-oriented instruction sets, designed to match a fast processor with a slow memory in programming environments where context switches are infrequent (1 in $\approx$ 25000 instructions), are not appropriate in a processing node containing a fast local memory and in an environment where context switches happen every 20 instructions.

The solution adopted in many machines is to increase the memory size of the node so a larger part of the problem can be performed in each node. This has the effect of reducing the concurrency to a point where the number of instructions executed between messages approaches 10^3. This increases the perceived efficiency from 20% to 90%. Efficiency is often measured in units of $\frac{T_P}{T}$ where T_P is time spent on useful work and T is total time. This measure of efficiency, however, ignores the cost of the node. If instead we measure efficiency in units of $\frac{T_P}{TA}$, where A is the total chip area of the node, the actual efficiency has been reduced by making the node larger. To truly increase the efficiency, we must build small, efficient nodes.

At MIT, we are developing the message-driven processor (MDP), a small, efficient processing node for a message-passing concurrent computer [10]. It is designed to support fine-grain concurrent programs by reducing the overhead and latency associated with receiving a message, by reducing the time necessary to perform a context switch, and by providing hardware support for object-oriented concurrent programming systems.

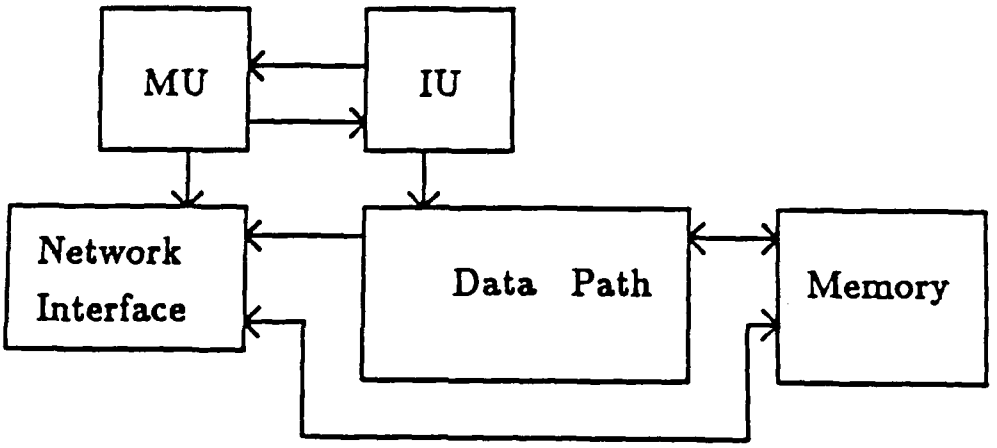


Figure 13: Block diagram of a message-driven processor (MDP). The message unit (MU) schedules messages, deciding whether to execute or buffer each arriving message. The instruction unit (IU), controls execution of instruction sequences used to carry out the functions requested by messages.

Message handling overhead is reduced by directly executing messages rather than interpreting them with sequences of instructions. As shown in Figure 13, the MDP contains two control units, the instruction unit (IU) that executes instructions and the message unit (MU) that executes messages. The MU performs message reception and task scheduling in hardware. When a message arrives it is examined by the MU which decides whether to queue the message or to execute the message by preempting the IU. Messages are enqueued without interrupting the IU. Message execution is accomplished by immediately vectoring the IU to the appropriate memory address. Special registers are dedicated to the MU so no time is wasted saving or restoring state when switching between message and instruction execution.

Context switch time is reduced by making the MDP a memory rather than register based processor. Each MDP instruction may read or write one word of memory. Because the MDP memory is on-chip, these memory references do not slow down instruction execution. Four general purpose registers are provided to allow instructions that require up to three operands to execute in a single cycle. The entire state of a context may be saved and restored in less than 12 clock cycles. Two register sets are provided, one for each of two priority levels, to allow low priority messages to be preempted without saving state.

An MDP word is 36-bits: a 4-bit tag and a 32-bit datum. Tags are used both to support dynamically-typed programming languages and to support concurrent programming constructs such as relocatable objects and futures.

The MDP is intended to support a fine-grain, object-oriented concurrent programming system in which a collection of objects interact by passing messages. In such a system, addresses are object names (identifiers). Execution is invoked by sending a message specifying a method to be performed, and possibly some arguments to an object. When an object receives a message it looks up and executes the corresponding method. Method execution may involve modifying the object's state, sending messages, and creating new objects. Because the messages are short (typically 6 words), and the methods are short (typically 20 instructions) it is critical that the overhead involved in receiving a message and in switching tasks to execute the method be kept to a minimum. In the MDP, the sum of these two overheads is less than 1 μs.

Rather than providing a large message set hard-wired into the MDP, we implement only a single primitive message, EXECUTE. This message takes as arguments a priority level (0 or 1), an opcode, and an optional list of arguments. The message opcode is a physical address to the routine that implements the message. More complex messages, such as those that invoke a method or dereference an identifier, can be implemented almost as efficiently using the EXECUTE message as they could if they were hard-wired. We choose not to implement complex messages in microcode because they will run just as fast using macrocode and implementing them in macrocode gives us more flexibility. Since the MDP is an experimental machine we place a high value on providing the flexibility to experiment with different concurrent programming models and different message sets, and to instrument the system. For example, we can patch the CALL firmware to measure the number of CALL messages received by each node.

The MDP memory can be accessed either by address or by content, as a set-associative cache. Cache access is used to provide address translation from object identifier to object location. This translation mechanism is used to support a global address space. Object identifiers in the MDP are global. They are translated at run time to find the node on which the object resides and the address within this node at which the object starts.

Because the MDP maintains a global name space, it is not necessary to keep a copy of the program code (and the operating system code) at each node. In fact, a copy of the entire operating system will not fit into a node's memory. Each MDP keeps a method cache in its memory and fetches methods from a single distributed copy of the program on cache misses.

A block diagram of the MDP memory is shown in Figure 14. The memory system consists of a memory array, a row decoder, a column multiplexor and comparators, and two row buffers (one for instruction fetch and one for queue access). Word sizes in this figure are for our prototype which will have only 1K words of RAM.

In the prototype, the memory array will be a 256-row by 144-column array of 3 transistor DRAM cells. In an industrial version of the chip, an 8K word (or larger) memory is feasible. We wanted to provide simultaneous memory access for data operations, instruction fetches, and queue inserts; however, to achieve high memory density we could not alter the basic memory cell. Making a dual port memory would double the area of the basic cell. Instead, we have provided two row buffers that cache one memory row (4 words) each. One buffer is used to hold the row from which instructions are being fetched. The other holds the row in which message words are being enqueued.

Some may argue that the MDP is unbalanced according to the rule of thumb stating that a 1MIP processor should have a 1MByte memory. The MDP is an $\approx$ 4MIP processor and only

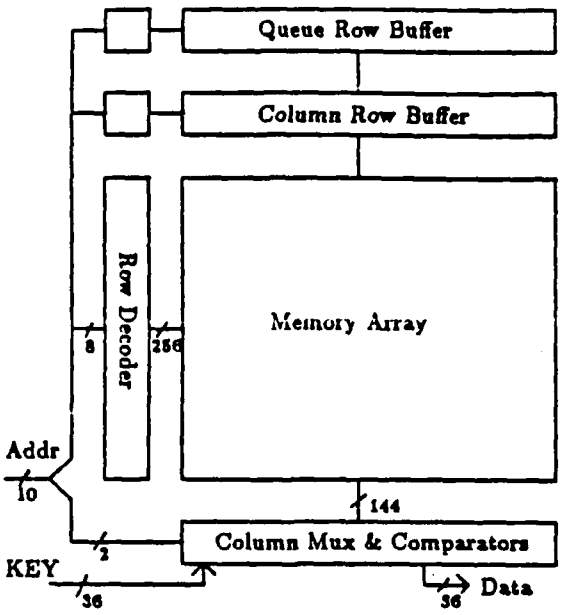


Figure 14: The MDP memory adds functionality in its peripheral circuitry while preserving the density of a simple memory array. Associative access is implemented by designing comparators into the column multiplexor. Row buffers for the message queue and instruction stream increase the effective memory bandwidth.

has a 36KByte memory. We argue however that it is not the size of the memory in a single node that is important, but rather the amount of memory that can be accessed in a given period of time. In a 64K node machine constructed from MDPs and using a fast routing network, a processor will be able to access a uniform address space of 2^{29} words (2^{31} Bytes) in less than $10\mu s$.

In a concurrent, object-oriented programming system, programs operate by sending messages to objects. Each method results in the execution of a method. The MDP supports this model of programming with the CALL and SEND messages. The execution sequence for a CALL message is shown in Figure 15. The first word of the message contains the priority level (0), and the physical address of the CALL subroutine. If the processor is idle, in the clock cycle following receipt of this word, the first instruction of the call routine is fetched. The CALL routine then reads the object identifier for the method. This identifier is translated into a physical address in a single clock cycle using the translation table in memory. If the translation misses, or if the method is not resident in memory, a trap routine performs the translation or fetches the method from a global data structure. Once the method code is found, the CALL routine jumps to this code. The method code may then read in arguments from the message queue. The argument object identifiers are translated to physical memory base/length pairs using the translate instruction. If the method needs space to store local state, it may create a context object. When the method has finished execution, or when it needs to wait for a reply, it executes a SUSPEND instruction passing control to the next message.

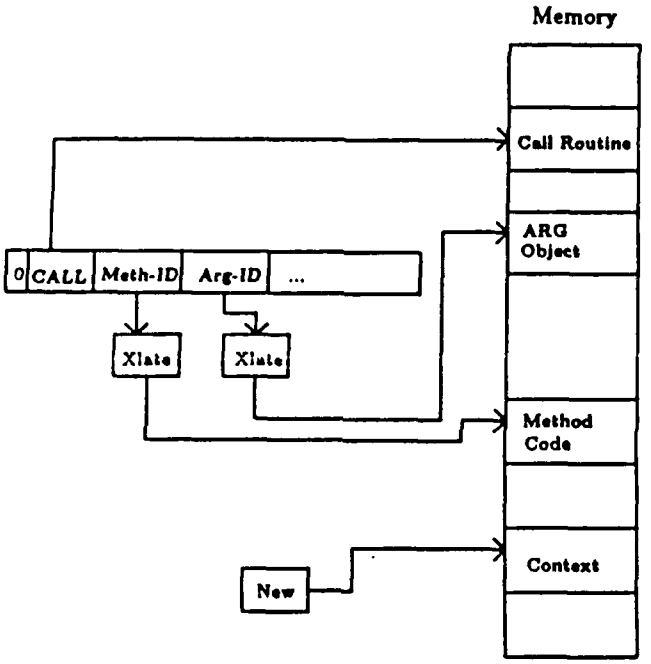


Figure 15: The CALL message invokes a method by translating the method identifier to find the code, creating a context (if necessary) to hold local state, and translating argument identifiers to locate arguments.

The MDP provides many of the advantages of both message-passing multicomputers and shared-memory multiprocessors. Like a shared-memory machine, it provides a single global name space, and needs to keep only a single copy of the application and operating system code. Like a message-passing machine, the MDP exploits locality in object placement, uses messages to trigger events, and gains efficiency by sending a single message through the network instead of sending multiple words. While we plan to implement an object-oriented programming system on the MDP, we also see the MDP as an emulator that can be used to experiment with other programming models.

5. NUMERICAL PROCESSING ELEMENTS

The performance of VLSI arithmetic chips is limited by the available I/O bandwidth, not by circuit speeds or circuit densities. With existing technology one can implement a full adder cell in $\approx 5K\lambda^2$ (λ is half the minimum line width [25]) area with a worst case delay of 1ns. Using such adders one can easily construct a 100MFLOPS pipelined floating-point adder/multiplier in an area of $32M\lambda^2$ which would fit comfortably on a single chip. The problem, however, is to transfer the 300 million 64-bit operands and results on and off the chip each second. The required bandwidth of $\approx$ *20Gbits/sec exceeds the capabilities of available packaging technologies.*

The I/O bandwidth of a 100MFLOPS arithmetic chip can be reduced to a manageable level of $\approx$ 2Gbits/sec by exploiting the locality inherent in the problems that use floating-point arithmetic. Most algorithms that make heavy use of floating-point do so in equations that involve between 10 and 50 arithmetic operations. Within each equation there is considerable locality. The result of one arithmetic operation feeds directly into the input of the next operation.

We can exploit this locality by constructing a number of small, slow floating point units and chaining them together using a reconfigurable switch. In the same area used to construct one 100MFLOPS pipelined unit we can construct 16, 6.25MFLOPS nibble-serial units. (In fact we can do considerably better since the nibble serial approach is more efficient.) Because the nibble-serial units are not pipelined and individually have narrow (4-bit wide) data paths they can be efficiently chained together to directly implement an equation. Without pipelining, the result of one operation is immediately available to be used as input during the next *operation cycle*. In a pipelined design, the result is not available for several operation cycles and other data must be found to keep the pipeline busy. The 4-bit wide data paths allow us to construct a switch that requires 2^{-8} times as much area as a 64-bit wide switch. This *equation chaining* is similar in some respects to the *vector chaining* performed by the vector pipelines of some supercomputers [28].

In addition to exploiting locality, equation chaining also allows us to efficiently exploit concurrency at the operation level. By statically scheduling all of the operations within an equation we avoid the overhead of synchronizing on each operation as is done in a dataflow machine [14].

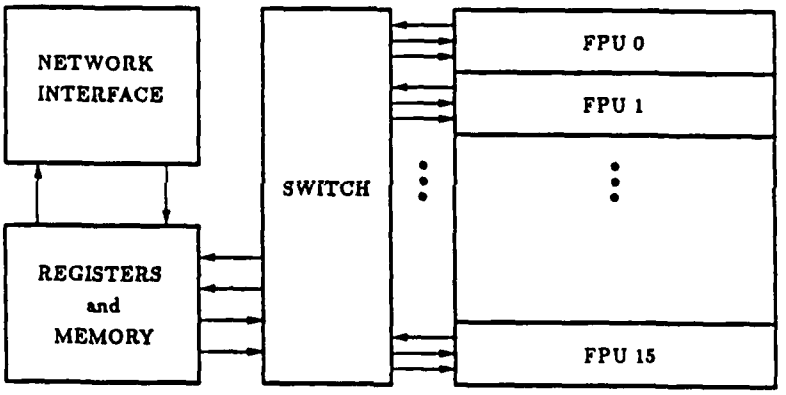


Figure 16: The Reconfigurable Arithmetic Processor (RAP) exploits the locality in equations to package 100MFLOPS on a chip while limiting I/O bandwidth to 10 % of that required by a conventional floating point processor. The RAP consists of a number ($\approx$ 16) nibble-serial floating point units connected by a statically reconfigurable switch. A register file and local memory are used to buffer problem instances.

Instead, we synchronize only once for the entire equation and then clock the data in lockstep through the function units as is done in a systolic array [24].

To test the idea of equation chaining we are developing a Reconfigurable Arithmetic Processor (RAP) chip [15]. In the RAP, 16 nibble-serial, floating-point function units are interconnected by a statically reconfigurable switching network. By changing the switch settings the chip can switch between different equations, or successive stages of a complex equation. As shown in Figure 16, the chip also includes a number of registers and a small amount of memory for buffering problem instances.

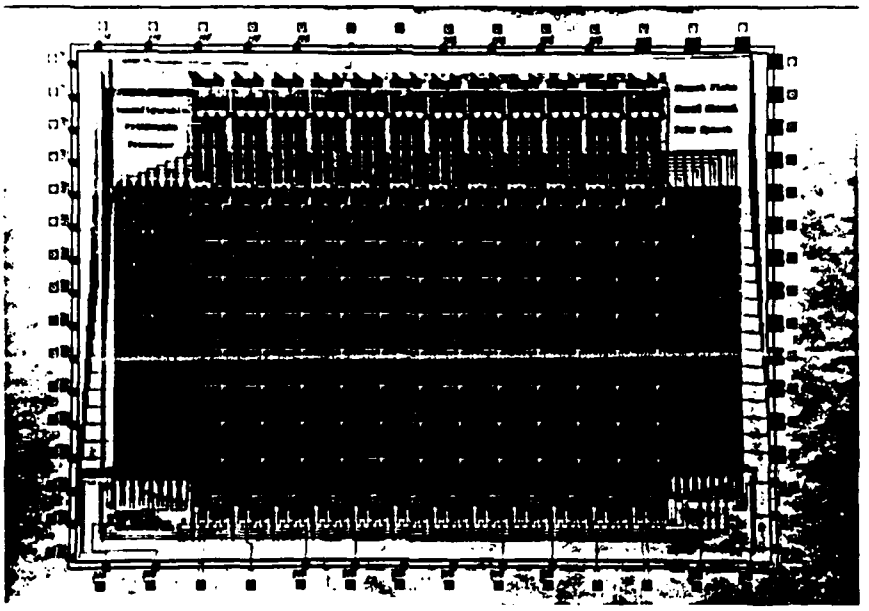


Figure 17: RAP test chip incorporates 12, 16-bit fixed point arithmetic units, a statically reconfigurable switch, and two register files.

The highest ratio of performance to area occurs when an arithmetic operation is broken down into a unit that can just be completed in a single cycle of the fastest clock that could be distributed to the unit, say 5ns. Most commercial floating-point chips (e.g., thos manufactured by Weitek) are implemented as a combinational function unit broken into two or three pipeline stages to improve throughput. Since the delay through each stage is much longer than the synchronization period for the chip, only a narrow *wavefront* of logic is active at any one time. The majority of the logic is idle.

To more efficiently use silicon area, the RAP uses nibble-serial arithmetic units. Because the carry paths in these units are shorter, we can clock them faster and keep more of the silicon busy at a given instant in time. While these units will take 16 clock periods to complete a single multiply operation, we can safely run them at clock rates of 100MHz giving a performance per unit of 6.25 MFLOPS.

A RAP test chip has been implemented to test several RAP components. Shown in Figure 17 this chip consists of 12, 16-bit di-bit serial, fixed-point arithmetic units connected by a statically reconfigurable sparse crossbar switch. Two register files store input and output operands and perform parallel-serial and serial-parallel conversion. This chip was designed by MIT students Stuart Fiske, Josef Shaoul, and Petr Spacek.

Consider the N-body problem described in Section 2. For each pair of bodies, b_i, and, b_j, one must compute the accelerations $a_{i,j}$ and $a_{j,i}$ using (1). The data flow graph for this calculation is shown in Figure 18. At compile time, the equation is translated to the data-flow graph. The graph is optimized to eliminate common subexpressions and to minimize the expression depth. The graph is then translated to a sequence of switch settings for the RAP. At runtime the RAP receives a message containing the equation's arguments and a continuation. The RAP then sequences the data through the function units to perform the calculation. The results are transmitted in a message as specified by the continuation.

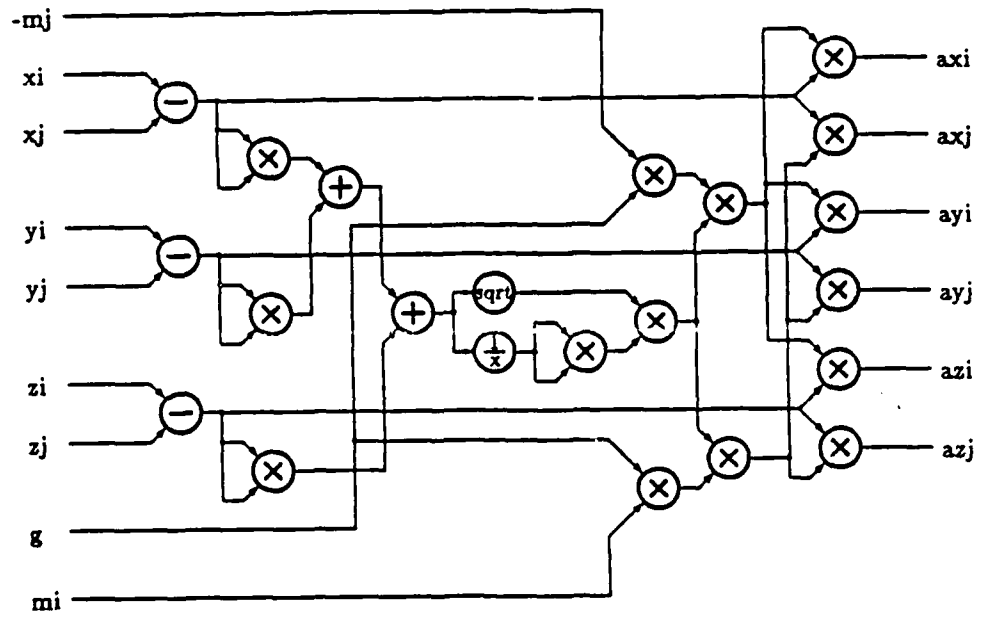


Figure 18: Data-flow graph for the acceleration calculation of the N-body problem. This calculation requires 22 arithmetic operations and 14 words of I/O. It can be performed on a RAP in 11 operation cycles giving an arithmetic rate of 37MFLOPS and an I/O bandwidth of 510Mbits/sec.

This computation requires 22 arithmetic operations: five add/subtracts, fifteen multiplies, one reciprocal, and one square root. It takes eight arguments and produces six results. Performing this operation using chained arithmetic units requires a total of 14 words of I/O compared with 64 words that would be required if the operation were performed using word parallel arithmetic units. This data flow graph has depth 9 and requires 11 operation cycles (the reciprocal and square root both require 3 cycles) on a RAP. With an operation rate of 6.25MHz the latency of the calculation is 1.8μs. The average concurrency is 5.9 operations/cycle (37MFLOPS) and the required I/O bandwidth is 81 bits/cycle, 510Mbits/sec. Synchronization overhead reduces the arithmetic rate and increases the I/O bandwidth slightly. On this problem the RAP achieves $\approx$37% of its peak performance with an I/O bandwidth that is well-matched to the capabilities of our interconnection network.

6. CONCLUSION

Recent advances in the design of interconnection networks, symbolic processors, and arithmetic processors have been described:

- Low-dimensional k-ary n-cube networks that use wormhole routing and virtual channels can send a 6-word message across the diameter of a 4K-node concurrent computer in 5μs. These low-dimensional networks ($8 \leq k \leq 64$ and $2 \leq n \leq 4$) outperform binary n-cubes ($k = 2$) because they balance the component of latency due to message length with the component due to distance. These networks are implemented with VLSI chips such as the TRC [7] and the NDF [11] that perform all routing and buffering internally using no memory bandwidth or CPU time on intermediate nodes.

- The Message-Driven Processor (MDP) can perform a task switch on message arrival in $1\mu s$. The MDP performs message reception, buffering, and scheduling in hardware to eliminate the software overhead of $100\mu s$ or more associated with these functions. Task switches are performed quickly because the MDP is memory rather than register based. The MDP memory provides both associative and indexed access. The associative access is used to support a global virtual address space needed to support concurrent programming systems.

- The Reconfigurable Arithmetic Processor (RAP) exploits the locality inherent in equations to achieve high arithmetic performance (100MFLOPS) with low I/O bandwidth. With today's VLSI technology, arithmetic performance is limited by I/O bandwidth rather than by the amount of logic that can be packaged on a single chip. The RAP executes equations asynchronously (in response to messages) and synchronously executes operations within an equation. This approach allows the RAP to exploit concurrency at the level of single arithmetic operations (e.g., $\times$ or $+$) without incurring a costly synchronization overhead on each operation.

In addition to improving performance, these developments are making concurrent computers easier to program:

- A global virtual address space in combination with a fast communication network allows programmers to operate on data independent of its location. Programmers need not be concerned with the physical topology of the machine they are using or with the mapping of their problem onto the nodes of the machine. Their only concern is with the logical structure of their problem.

- A processor that supports a fast task switch in reponse to a message allows programs to be broken into extremely fine grains (as small as 20 instructions) without loss of efficiency. This permits programmers to decompose a program into its *natural* units: objects (e.g., particles) and methods (e.g., force calculation).

- Message passing provides a simple and powerful synchronization mechanism. Programmers need not be concerned with locks, semaphores, or monitors. Operations are scheduled by the arrival of the message(s) containing the required data.

Concurrent programming is not difficult if suitable abstractions are used. Programmers should use the natural partition of the problem and not be concerned with placement. Synchronization can be performed by allowing the data flow of the program to sequence the required operations. As this technology matures, we expect to see abstractions for concurrency that will make concurrent programming no more difficult than sequential programming.

This paper has drawn on examples from the J-Machine, a message-passing concurrent computer under development at MIT. We expect to complete the construction of a prototype J-Machine in 1990. The status of the project at the time of this writing (June 1987) is as follows:

- the design of the NDF is complete, and a test chip (no node logic) is being fabricated;

- a RAP simulator is complete, and a RAP test chip (Figure 17) is being fabricated;

- instruction-level and register-transfer-level simulators for the MDP are complete, and portions of the datapaths and memory have been laid out;

Many challenging problems in the design of hardware and software for concurrent computers remain. A major research area is the design of fault tolerant systems. While we can construct a 4K node machine with an MTBF of 2400 hours (4K chips at 100FITS), future machines may have MTBFs of only a few hours and will require architectures that can survive node and link failures without loss of data.

Concurrent software systems are still quite primitive. Abstractions for concurrency that express common patterns of computation while hiding the details of implementation are required. Compilers should perform optimizations that expose concurrency in programs and automate the placement of objects onto processing nodes. To exploit chips like the RAP, compilers that map equations onto function units are needed. We also need compile-time and run-time resource management technolgy to regulate the concurrency in programs to match the available computing resources. Concurrent software technology must mature for these powerful machines to see widespread use.

ACKNOWLEDGEMENTS

The following MIT students have contributed to the work described here: Linda Chao, Andrew Chien, Stuart Fiske, Soha Hassoun, Waldemar Horwat, Jon Kaplan, Michael Larivee, Josef Shaoul, Paul Song, Petr Spacek, Brian Totty, and Scott Wills.

I thank Carol Roberts and Glen Ring for their assistance in typesetting this document.

I thank Tom Knight, Gerry Sussman, Steve Ward, Dave Gifford, and Carl Hewitt of MIT, and Chuck Seitz and Bill Athas of Caltech for many valuable suggestions, comments, and advice.

REFERENCES

[1] Agha, Gul A., *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, MA, 1986.

[2] Athas, W.C., and Seitz, C.L., *Cantor Language Report*, Technical Report 5232:TR:86, Dept. of Computer Science, California Institute of Technology, 1986.

[3] BBN Advanced Computers, Inc., *Butterfly Parallel Processor Overview*, BBN Report No. 6148, March 1986.

[4] Bryant, R., "A Switch-Level Model and Simulator for MOS Digital Systems," *IEEE Transactions on Computers*, Vol. C-33, No. 2, Feb 1984, pp. 160-177.

[5] Dally, William J., *The Balanced Cube: A Concurrent Data Structure*, Technical Report 5174:TR:85, Dept. of Computer Science, California Institute of Technology, 1985.

[6] Dally, William J., *A VLSI Architecture for Concurrent Data Structures*, Kluwer, Hingham, MA, 1987.

[7] Dally, William J. and Seitz, Charles L., "The Torus Routing Chip," *J. Distributed Systems*, Vol. 1, No. 3, 1986, pp. 187-196.

[8] Dally, William J. "Wire Efficient VLSI Multiprocessor Communication Networks," *Proceedings Stanford Conference on Advanced Research in VLSI*, Paul Losleben, Ed., MIT Press, Cambridge, MA, March 1987, pp. 391-415.

[9] Dally, William J. and Seitz, Charles L., " Deadlock-Free Message Routing in Multiprocessor Interconnection Networks," *IEEE Transactions on Computers*, Vol. C-36, No. 5, May 1987, pp. 547-553.

[10] Dally, William J. et.al., "Architecture of a Message-Driven Processor," *Proceedings of the 14th ACM/IEEE Symposium on Computer Architecture*, June 1987, pp. 189-196..

[11] Dally, William J., and Song, Paul., "Design of a Self-Timed VLSI Multicomputer Communication Controller," To appear in, *Proc. IEEE International Conference on Computer Design*, 1987.

[12] Dally, William J., "Concurrent Data Structures," Chapter 7 in *Message-Passing Concurrent Computers: Their Architecture and Programming*, C.L. Seitz et. al., Addison-Wesley, Reading, MA, publication expected 1987.

[13] Dally, William J., "The J-Machine: A Concurrent VLSI Message-Passing Computer for Symbolic and Numeric Processing," to appear.

[14] Dennis, Jack B., "Data Flow Supercomputers," *IEEE Computer*, Vol. 13, No. 11, Nov. 1980, pp. 48-56.

[15] Fiske, Stuart, and Dally, William J., "The Reconfigurable Arithmetic Processor," to appear.

[16] Goldberg, Adele, and Robson, David, *Smalltalk-80, The Language and Its Implementation*, Addison-Wesley, Reading, Mass., 1983.

[17] Halstead, Robert H., "Parallel Symbolic Computation," *IEEE Computer*, Vol. 19, No. 8, Aug. 1986, pp. 35-43.

[18] Intel Scientific Computers, *iPSC User's Guide*, Order No. 175455-001, Santa Clara, CA, Aug. 1985.

[19] Kermani, Parviz and Kleinrock, Leonard, "Virtual Cut-Through: A New Computer Communication Switching Technique," *Computer Networks*, Vol 3., 1979, pp. 267-286.

[20] Kernighan Brian, W., and Ritchie, Dennis M., *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ., 1978.

[21] Kirkpatrick, S., Gelatt, C.D. Jr., and Vecchi, M. P., "Optimization by Simulated Annealing," *Science*, Vol. 220, No. 4598, 13 May 1983, pp. 671-680.

[22] Knight, Tom, and Krymm, Alex, "Self Terminating Low-Voltage Swing CMOS Output Driver," *Proc. Custom Integrated Circuits Conference*, 1987.

[23] Kuck, David J., Muraoka, Yoichi, and Chen, Shyh-Ching, "On the Number of Operations Simultaneously Executable in Fortran-Like Programs and Their Resulting Speedup," *IEEE Transactions on Computers*, Vol. C-21, No. 5, May 1972, pp. 1293-1310.

[24] Kung, H.T., "Why Systolic Architectures," *IEEE Computer*, Vol. 15, No. 1, Jan. 1986, pp. 37-46.

[25] Mead, Carver A. and Conway, Lynn A., *Introduction to VLSI Systems*, Addison-Wesley, Reading, Mass., 1980.

[26] Nicolau, Alexandru and Fisher, Joseph A., "Measuring the Parallelism Available for Very Long Instruction Word Architectures," *IEEE Transactions on Computers*, Vol. C-33, No. 11, Nov. 1984, pp. 968–976.

[27] Press, William H., Flannery, Brian P., Teukolsky, Saul A., and Vetterling, William T., *Numerical Recipes*, Cambridge University Press, New York, 1986, Ch. 15.

[28] Russel, Richard M., "The CRAY-1 Computer System," *Comm. ACM*, Vol. 21, No. 1, Jan. 1978, pp. 63-72.

[29] Seitz, Charles L., "System Timing" in *Introduction to VLSI Systems*, C. A. Mead and L. A. Conway, Addison-Wesley, 1980, Ch. 7.

[30] Seitz, Charles L., et al., *The Hypercube Communications Chip*, Display File 5182:DF:85, Dept. of Computer Science, California Institute of Technology, March 1985.

[31] Seitz, Charles L., "The Cosmic Cube", *Comm. ACM*, Vol. 28, No. 1, Jan. 1985, pp. 22-33.

[32] Seitz, Charles L., Athas, William C., Dally, William J., Faucette, Reese, Martin, Alain J. , Mattisson, Sven, Steele, Craig S., and Su, Wen-King, *Message-Passing Concurrent Computers: Their Architecture and Programming*, Addison-Wesley, publication expected 1987.

[33] Steele, Craig S., *Placement of Communicating Processes on Multiprocessor Networks*, Technical Report 5184:TR:85, Dept. of Computer Science, California Institute of Technology, 1985.

[34] Su, Wen-King, Faucette, Reese, and Seitz, Charles L., *C Programmer's Guide to the Cosmic Cube*, Technical Report 5203:TR:85, Dept. of Computer Science, California Institute of Technology, September 1985.

[35] Wirth, N., and Jensen, K., *Pascal User Manual and Report*, Springer-Verlag, New York, 1975.

END

DATE

FILMED

8-88

DTIC